RL-TR-95-184
Final Technical Report
October 1995

HIERARCHICAL MODELING AND SIMULATION SYSTEM (HI-MASS)

Syracuse University

Robert G. Sargent and Douglas G. Fritz

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-184 has been reviewed and is approved for publication.

APPROVED:

ALEX F. SISTI
Project Engineer

FOR THE COMMANDER:

DELBERT B. ATKINSON, Colonel, USAF
Director of Intelligence & Reconnaissance

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (IRAE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1995	Final May 94 - Mar 95

4. TITLE AND SUBTITLE

HIERARCHICAL MODELING AND SIMULATION SYSTEM (HI-MASS)

5. FUNDING NUMBERS

C - F30602-94-C-0098
PE - 62702F
PR - 4594
TA - 15
WU - L4

6. AUTHOR(S)

Robert G. Sargent and Douglas G. Fritz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Syracuse University
439 Link Hall
Syracuse NY 13244

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/IRAE
32 Hangar Rd
Rome NY 13441-4114

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-95-184

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Alex F. Sisti/IRAE/(315) 330-4518

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objectives of this contract were to demonstrate the feasibility, and evaluate the useability, of Hierarchical Control Flow Graph (HCFG) Models as a way of specifying hierarchical models for discrete event simulation. The scope of the contract was to develop a prototype system called HI-MASS (Hierarchical Modeling and Simulation System) to aid in accomplishing the objectives of the contract and to evaluate the effectiveness of hierarchical modeling using HCFG Models for discrete event simulation.

14. SUBJECT TERMS

Hierarchical simulation, Control flow graphs (CFG), Hierarchical control flow graph (HCFG), Atomic omponent (AC)

15. NUMBER OF PAGES

40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Introduction

This is the final scientific and technical report for Contract F30602-94-C-0098 from Rome Laboratory which covered the period of May 1994 to March 1995. The objectives of this contract were to demonstrate the feasibility of and evaluate the useability of Hierarchical Control Flow Graph (HCFG) Models [FS93] as a way of specifying hierarchical models for discrete event simulation. The scope of the contract was to develop a prototype system called HI-MASS (Hierarchical Modeling and Simulation System) to aid in accomplishing the objectives of the contract and to evaluate the effectiveness of hierarchical modeling using HCFG Models for discrete event simulation.

An overview of HCFG Models is given in Section 2 of this report. Also briefly described are Control Flow Graph (CFG) Models [CS90a] on which HCFG Models are based. CFG Models were developed as a representation (specification) language to obtain "automatic lookahead" for parallel/distributed discrete event simulation algorithms [CS89, CS90b, CS90c, CS90d] in order to avoid a modeler having to add "lookahead" information as is the common practice. HCFG Models were developed as a hierarchical modeling (specification) language that allowed CFG Models to be used as a representation language. See Figure 1.

Section 3 contains an overview of HI-MASS. HI-MASS is a prototype simulation system that provides for specification of HCFG Models and uses a sequential synchronous simulation algorithm. HI-MASS is a C++ based system specifically designed to operate on a SUN4 SPARC Workstation running SunOS-4.1.3. However, HI-MASS (excluding the GUI) has also been run on other Unix based workstations including an IBM RS/6000 running AIX, a DEC Alpha running OSF/1, and an Intel 486 based notebook computer running Linux.

Presented in Section 4 are the lessons learned in performing this work and the evaluations of HCFG Models for performing discrete event simulation and of HI-MASS. Section 5 contains the recommendations and summary.

2 Hierarchical Control Flow Graph Models

Hierarchical Control Flow Graph (HCFG) Models [FS93] is a hierarchical modeling paradigm for discrete event simulation. HCFG Models are based on and designed to be an extension of Control Flow Graph (CFG) Models [CS90a]. HCFG Models add hierarchical modeling capability to CFG Models to aid in the modeling of more complex systems while maintaining the basic model representation properties of CFG Models. HCFG Models can be algorithmically mapped (flattened) into equivalent CFG Models[FS93]. HCFG Models are also a true superset of CFG models in that any valid CFG Model is also a valid HCFG Model.

Cota and Sargent developed the Control Flow Graph (CFG) Model representation [CS90a] based on the modified process interaction world view [CS92] in which encapsulated model components interact via message passing. The primary objective of CFG Models was to make information useful for parallel simulation explicit in the model representation. Cota and Sargent then developed a set of algorithms [CS90c] for the execution of CFG Models that allow CFG Models to be executed on either sequential architectures or parallel/distributed architectures without any additional modeler input. The parallel algorithms for CFG Models generate "automatic lookahead" information which is a key element in parallel simula-

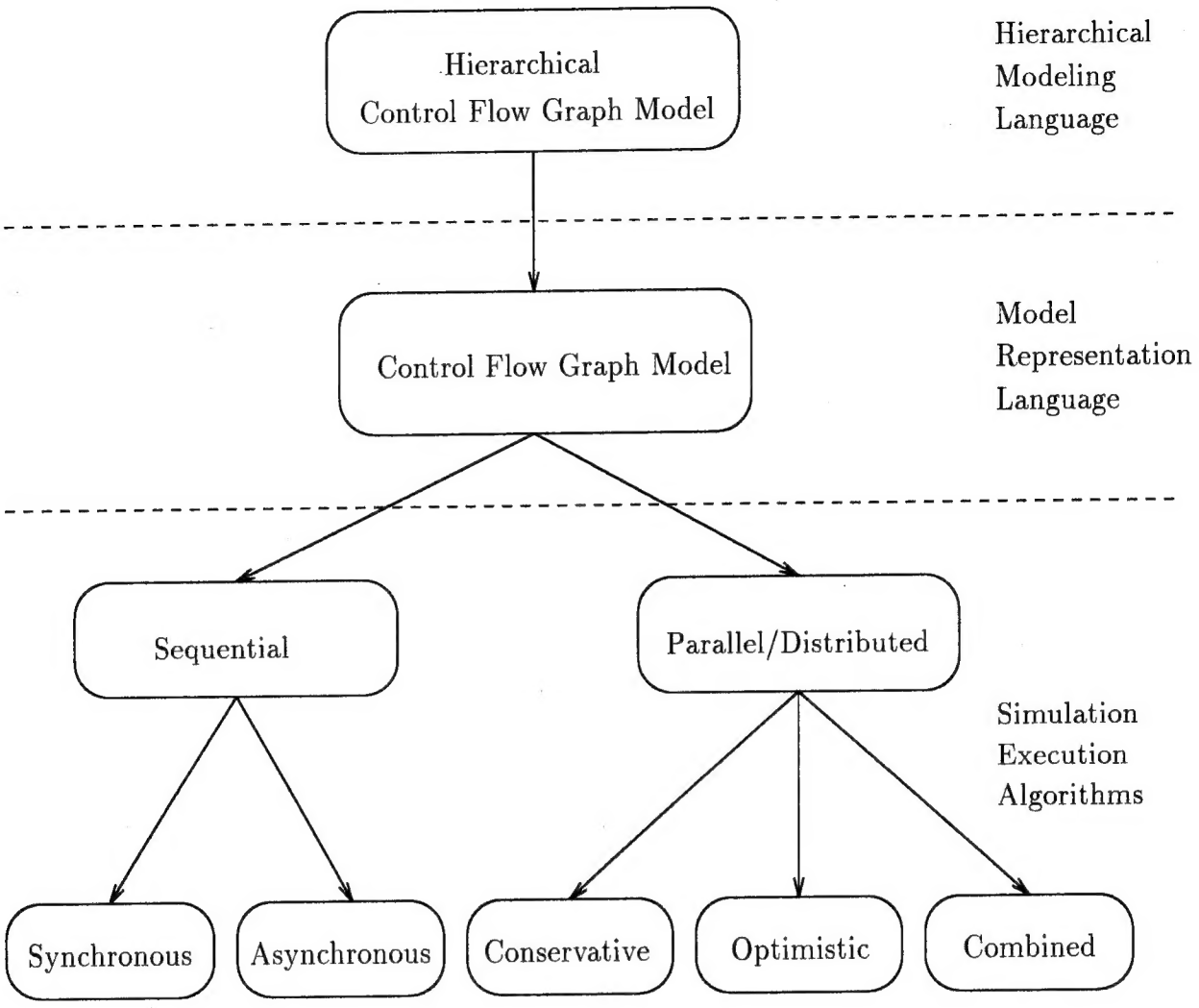

Figure 1: Languages and Algorithms

tion [Fuj90]. The ability to use these existing algorithms for model execution is a desirable property for a model representation.

A key property of CFG models is the "modularity" of model components. The modularity property is defined to include the both the "locality" and "encapsulation" properties. Modularity of components has certain advantages, such as those given in [CS92]:

1. Modularity makes modeling of complex systems easier by allowing a modeler to focus on the structure of one part of the system at a time.
2. Modularity makes modification of the model easier. It is possible to change one component in a model without determining the effects of that change on the rest of the model. This facilitates maintenance of the model over time and can also facilitate experimentation with alternative system designs by making

it easier to change aspects of the behavior of individual components of the system.

3. Modularity is required to support a hierarchical approach to simulation modeling, such as that discussed in [Zei84]. (However, modularity is not the only thing required for hierarchically structured modeling. As discussed in [Zei84], there must also be some mechanism by which components in a model can be *coupled.*)
4. Modularity is required to support reuse of a model component in different simulation models. In order to define a model component that can be reused in different models, it is necessary to define the component without reference to other components in the model.
5. Encapsulation may make it easier to develop a parallel simulation based on a given model. As discussed in [Fuj90], a parallel discrete event simulation is carried out by having a number of *logical processes*[model components] each keep track of the state of different components of the system. Each logical process uses a separate simulation clock, and the value of different simulation clocks may be different at any point during the simulation. For this reason, a logical process cannot, in general, correctly access the state of a component maintained by another logical process. Instead, the logical processes must communicate by sending messages, as in [Jef85] and [Mis86], or by accessing shared variables, as in [Rey82].

A CFG Model consists of a set of interacting encapsulated Atomic Components (AC's) which communicate solely via message passing over unidirectional intercomponent message channels. All the AC's in a CFG Model are conceptually autonomous concurrently operating entities.

The AC's in a CFG Model use what is known as "active receiver" message passing which differs from the "passive receiver" message passing commonly used for object-oriented simulation. In active receiver message passing incoming messages are queued until the receiving AC chooses to handle them instead of being acted on immediately upon arrival as is done in the passive receiver case.

In the CFG Model representation, the behavior of each system component (or process) is specified by a Control Flow Graph. The interactions between components are accomplished via message passing over a set of directed channels which interconnect model (atomic) components and specify the static routing of all intercomponent message traffic. Each channel generally carries only one type of message which implies that there may be multiple channels between two components, each of which carries a different type of message. Messages leave a component via an output port and enter a component via an input port. Each channel connects exactly one output port to one input port and each port is connected to exactly one channel. Messages queue on input ports until the Control Flow Graph describing the behavior of the receiving component decides to receive them, i.e., components in CFG Models are active receivers. The timestamps used on intercomponent messages are the time at which the messages were sent. (This is in contrast to the method generally used in parallel and distributed simulation in which the timestamps specify the time at which the messages are to be

received.) The specification of channels is accomplished via an Interconnection Graph. An Interconnection Graph is a directed graph where the nodes represent the model atomic components and the directed edges represent the channels along which intercomponent messages flow.

Thus, the complete specification of a CFG Model requires two distinct types of specifications. The first type of specification, the Interconnection Graph (IG), specifies the set of components in the model and the static intercomponent message routing pattern for the model. Each CFG Model has exactly one IG. The second type of specification, a Control Flow Graph (CFG), is used to specify the behavior of the individual AC's in the model. Each type of AC in a model has its own CFG which defines the behavior for all AC's of that type in the model. If there are n different types of AC's in a particular CFG Model, then n CFG's (plus the IG) are required to completely specify that model.

HCFG Models provide hierarchical extensions to both types of specifications used in CFG Models. A Hierarchical Interconnection Graph (HIG) is a hierarchical extension of the CFG Model IG which allows the modeler to specify model components hierarchically by supporting the concept of "coupling" together existing model components to form new model components. A Hierarchical Control Flow (HCFG) is a hierarchical extension of CFG's which allows the modeler to use hierarchy in the specification of the behavior of an AC type by supporting the recursive partitioning of the component's behavioral state space.

The component and interconnection specification (the HIG of an HCFG Model) is discussed in the next subsection, and the AC behavior specification (HCFG's) is discussed in the following subsection.

2.1 Component and Interconnection Specification

In this subsection we discuss the component and interconnection specification of a model which specifies the set of components which make up the model and how those components are interconnected. We describe the HIG which is used to specify the components and interconnections for an HCFG Model and we also note the special case of a HIG which corresponds to the IG used in CFG Models.

In a CFG Model we develop a model as a set of interacting AC's. In an HCFG Model we introduce a new class of components called "Coupled Components" which are used in conjunction with AC's for the development of hierarchical models. A Coupled Component (CC) is a component which defines an encapsulated coupling of a set of subcomponents. These subcomponents may be either AC's and/or other CC's. Coupled components support the development of hierarchical models using "top down" recursive decomposition of components into smaller and simpler components, "bottom up" composition of existing components to form new components, or a combination of these two methods.

Each component in an HCFG model (whether atomic or coupled) is an encapsulated entity that has a name, a type, a set of output ports through which the component sends messages, and a set of input ports through which the component receives messages. A component's name is the "name" of a specific instance of a component type. Each instance has its own name whereas all components of the same type share the same "type" or "type name". For example, suppose we have a component type with a type name "Car". We may

have four Cars (instances of type Car) in our model with the names "aRedCar", "aBlueCar", "myCar", and "yourCar", all of which are instances of type "Car". Each Car has its own identity, but all Cars share the same definition.

Figure 2 shows the definition of a coupled component of type "MyModel" (i.e., its coupled component specification). A component of type "MyModel" contains two subcomponents, component "sam" of type "A" and component "bill" of type "B". (When both a component's name and its type are shown in the same figure we distinguish the two by placing the component's type inside parentheses.) A Component of type "A" has an input port named "ok" an output port named "out". A Component of type "B" has an input port named "in" and an output port named "go".

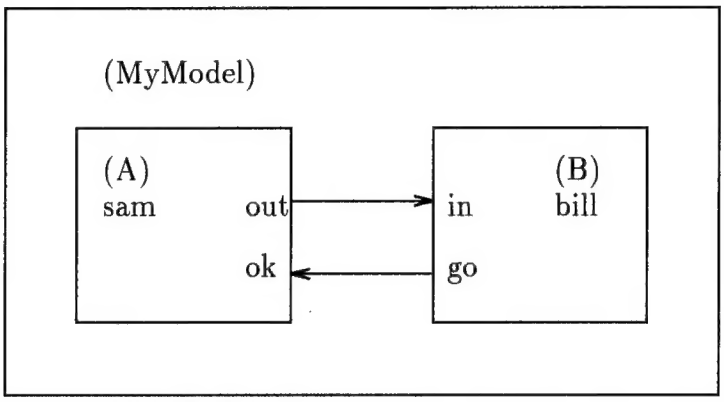

Figure 2: Coupled Component type "MyModel"

Component type MyModel (Figure 2) has no ports with which to communicate with the outside world. Each HCFG Model has exactly one coupled component which has this property, the coupled component which encloses the entire HCFG Model. If both component types "(A)" and "(B)" shown in Figure 2 are AC's (i.e., they have behavior specifications (HCFG's) and not subcomponents and coupling specifications) then Figure 2 is a special case of a coupled component which corresponds to an IG. The internal view of a coupled component defines a set of subcomponents and a coupling for a specific type of component. Thus, there is never an instance name shown for the enclosing coupled component (e.g., "(MyModel)" in Figure 2).

Now suppose that component type "B" shown in Figure 2 is actually a coupled component which contains three subcomponents with intercomponent coupling as shown in Figure 3. Note that from the internal view of a coupled component (the coupled component specification) only a component type is shown. There may be multiple components (instances) of the same component type in a model which all share the same type definition as is the case here where there are two instances (d0 and d1) of component type D.

The set of components of each HCFG Model can be viewed as a rooted tree structure which we call the HIG tree. The "top" or "root" component of the HIG tree for a model encloses the entire model. This top level component of the HIG tree is the only component in an HCFG model which has no ports. (By definition, there are no other components

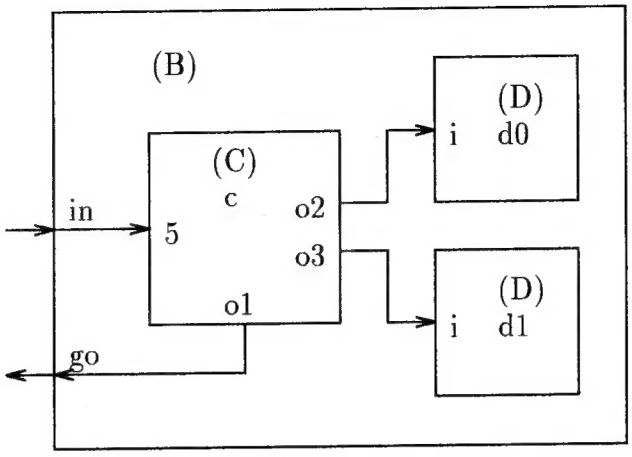

Figure 3: Coupled Component type "B"

outside the top level component for it to communicate with.) Also, note that all internal (non-leaf) nodes of the HIG tree represent coupled components and all leaf nodes in the HIG tree represent atomic components. The HIG tree shows the hierarchical relationship of the components in a model but not the intercomponent coupling information.

If we assume that all component types in our example model (Figures 2 and 3) other than "MyModel" and "B" are AC's, then the HIG tree for our model would be as shown in Figure 4. Note that all leaf nodes are AC's and the internal nodes are coupled components. There can be as many levels in a HIG as a modeler chooses to have.

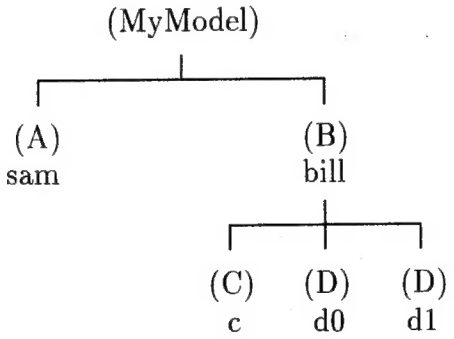

Figure 4: HIG Tree

A complete HIG specification for a model consists of a set of component specifications, one for each type of coupled component in the model. The HIG tree can be automatically generated from the set of CC specifications. A HIG can also be algorithmically flattened into an IG [FS93]. The IG which results from from flattening a HIG consists of a modified root node, no internal nodes, and all the leaf nodes of original HIG. All the intermediate

nodes (coupled components) are removed, while preserving the coupling information. The IG for our example in this subsection is shown in Figure 5. Notice that component "bill" of type "B" has been removed and replaced with its internal representation. (See [FS93] and [FDS95] for further discussion on mapping (flattening) a HIG into an IG.)

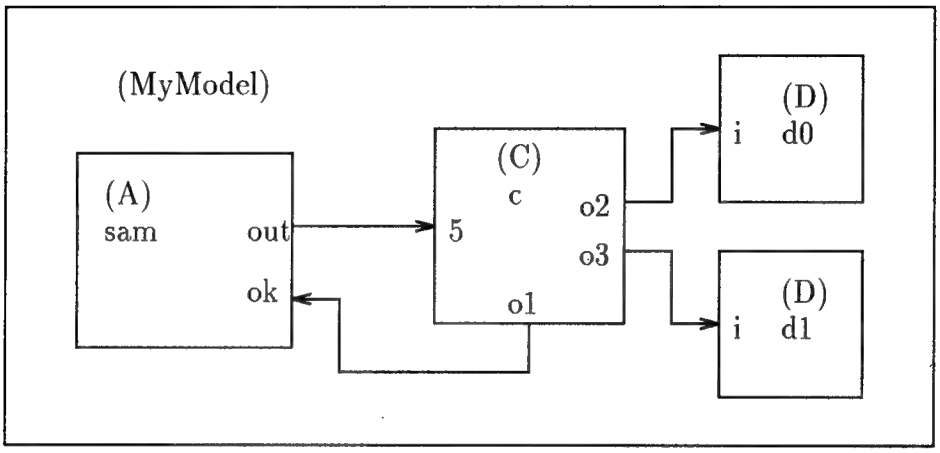

Figure 5: Interconnection Graph

2.2 Atomic Component Behavior Specification

In an HCFG Model the behavior definition for each type of Atomic Component (AC) is specified via an HCFG [FS93]. An HCFG is an hierarchical extension of a Control Flow Graph (CFG) [CS90a]. CFG's are a new way (world view) of specifying the behavior of an AC. In a CFG, the nodes represent control states of an AC. Control states are a formalization of the "process reactivation point" [CS92, Zei84] and can also be viewed as a partition of the (behavioral) state space of an AC. In an HCFG, Macro Control States (MCS's) [FS93] are used in addition to control states to specify the behavior of an AC. MCS's are an encapsulated set of control states and/or other MCS's. They can be viewed as a way to recursively partition an AC's behavioral state space into a set of disjoint partial behavior specifications.

2.2.1 Control Flow Graphs (CFG's)

In a CFG Model [CS90a] (also see [CS89, CS90b, CS90c, CS90d, CS92]) the behavior of an AC is specified using an augmented directed graph. A CFG consists of a set of nodes (Control States), a set of directed edges which interconnect the nodes (showing the possible control state transitions), a set of variables (local to the AC), and a set of functions (again, local to the AC). (Edges may originate and terminate on the same node.)

Each edge in a CFG also has three additional attributes: a priority, a condition, and an event. The priority of each edge is specified as a positive integer value. All edges originating from the same control state must have unique priorities. An edge with priority value of

one (1) is the highest priority edge leaving a specific control state (priority values of larger numbers indicate lower priority edges).

The condition attribute associated with each edge is of one of the following three types: "non-empty input port", "boolean expression", or "time delay". Edges that have condition attributes of these types are called "PortEdges", "BoolEdges", and "TimeEdges", respectively.

A PortEdge has an associated input port of that AC which is used to determine the status of the PortEdge's condition. A PortEdge's condition attribute is **True** now (*now* is the value of the AC's local simulation clock at the time the edge condition is evaluated) if there exists an unreceived message on the associated input port and **False** otherwise (i.e., as far as the simulation algorithm is concerned, the condition will *never* be **True**).

A BoolEdge has an associated condition function based on values of the the local variables of that AC which specifies a predicate which is tested whenever the BoolEdge's condition is evaluated. A BoolEdge's condition attribute is **True** now if the associated predicate function returns a value of **True**, and **False** otherwise.

A TimeEdge has an associated time delay function which specifies how far into the future we must wait until the TimeEdge's condition becomes **True**. A TimeEdge has an associated time delay function that returns a time delay value Δt. The TimeEdge's condition is **False** until (now $+\Delta t$), and **True** thereafter.

The event attribute associated with each edge specifies the action that is taken whenever that edge is traversed during the course of simulation execution. The action taken may include changing the values of local variables, sending messages to output ports, or reading a message from an input port if the event is associated with a PortEdge. Any event may send messages to one or more output ports, however only events that are associated with PortEdges may receive messages from input ports. An event associated with a PortEdge receives only one message from an input port during each execution of that event routine, and the message may only be received from the input port which is associated with that PortEdge.

Each AC has what is referred to as a "Point Of Control" (POC). The POC for an AC always resides at a Control State (CS). The control state at which the POC currently resides is called the "current" CS. The operation of an AC is as follows. The AC examines all edges leaving the current CS and selects the edge whose condition attribute will first (at the earliest point from the current simulation time) assume the value **True**. If this selection process returns more than one edge, then the edge with the highest priority (lowest number) is selected for traversal. To execute its next simulation event, the AC advances its local simulation clock (if necessary) to the time at which the selected edge's condition attribute becomes **True**. The AC's POC then traverses the selected edge to the control state which the edge terminates on, executing the associated event routine during the edge traversal. The control state which the POC arrives on during the edge traversal now becomes the current CS and the process of selecting an outbound edge begins again.

Cota and Sargent [CS89, CS90b, CS90c, CS90d, CS92] have developed different simulation execution algorithms for the Control Flow Graphs representation. Two simulation execution algorithms are available for sequential computers [CS90c] and [FS93]: synchronous and asynchronous. The synchronous algorithm uses the standard approach of computing the events

across all control flow graphs in their time order sequence. The asynchronous algorithm allows events to be executed out of time sequence when this does not effect the simulation results in order to reduce the simulation execution time by eliminating some event list operations. For parallel and distributed computers there are two conservative algorithms (one using null message passing and the other using deadlock detection and resolution), an optimistic algorithm which is more efficient than the usual optimistic algorithm because rollback is based upon the control states of a Control Flow Graph instead of rolling back if a "late message" is received to a Control Flow Graph, and an optimistic Only When Necessary (OWN) algorithm which is a combined optimistic and conservative algorithm. All of these algorithms obtain the information they need directly from Control Flow Graphs and thus a modeler need not add any additional information such as the "lookahead" information required for most parallel and distributed simulation execution algorithms. This is because lookahead is "automatically" obtained by the algorithms from the Control Flow Graphs representation.

As the complexity of an AC's behavior increases, it becomes more difficult to model that behavior due to the explosion of the number of control states and edges required. Hierarchical Control Flow Graphs allow the user to partition an AC's behavior into a disjoint set of partial behavior specifications called Macro Control States (MCS's). This partitioning of an AC's behavior can be done recursively until the remaining partial behavior specifications can be easily modeled using CFG's. This recursive partitioning of the behavior specification forms a tree structure called an HCFG tree in which the nodes of the tree are MCS's. HCFG's can be "flattened" into CFG's by using an algorithm [FS93].

2.2.2 Macro Control States (MCS's)

An MCS is an encapsulated entity used to specify a partial behavior of an AC. Each MCS specification contains the following. (One or more of the following sets for a specific MCS may be empty.)

- a set of input pins through which the POC can enter the MCS.
- a set of output pins through which the POC can exit the MCS
- a set of child MCS's which farther partition the AC's behavior as specified in the current MCS
- a set of control states
- a set of edges
- a set of member variables (members of the MCS) which may include variables local to the MCS and also handles to external information such as
    - input ports on which messages can arrive (of that AC)
    - output ports to which messages can be sent (of that AC)
    - variables which are members of another MCS (of that AC) or of the AC itself
    - functions which are defined in other MCS's or in the AC
- a set of condition functions (boolean and/or time delay)
- a set of event routines

The operation of an HCFG is an extension of the operation of a CFG. The POC for an AC resides at a control state called the current control state. This current control state is contained within a specific MCS called the current MCS. Edges are selected in the same manner as in CFG's. The POC leaves the current control state over the selected edge and the action specified by the event routine associated with the traversed edge is carried out just as in CFG's. However, in an HCFG, the selected edge may terminate on either a control state within the same MCS, on an input pin of a child MCS, or on an output pin of the current MCS. If the selected edge terminates on a control state within the current MCS then the operation is identical to that of CFG's. If the selected edge terminates on an input pin of a child MCS, then the POC enters that child MCS through the input pin. If the selected edge terminates on an output pin of the current MCS, then the POC leaves the current MCS and enters the current MCS's parent MCS in the HCFG tree via the current MCS's output pin.

When the POC traverses an edge that terminates on a pin, the POC will then continue to traverse one or more directed edges, starting with the edge leaving the pin, until it eventually arrives at a control state. In contrast to control states which may have an arbitrary number of outbound edges, each pin has exactly one outbound edge, thus no edge selection algorithm is required for edges leaving pins. Also, edges which originate from pins do not have the set of three attributes (priority, condition, and event) that edges originating from control states do. Thus there is never a condition test required before traversing an edge originating from a pin, and there is no associated event to be executed during the traversal.

An example of the use of CFG's and HCFG's in the specification of an AC's behavior is shown in the next subsection.

2.2.3 Hierarchical Control Flow Graph (HCFG) Example

To demonstrate how an HCFG is used to model the behavior of an AC, we show how to model the behavior of a simple queueing server. This server handles two classes of jobs using what we call a priority preempt/resume job selection discipline. Each job is either a "high priority" job, or a "low priority" job. Jobs within each class are processed on a First Come First Serve (FCFS) basis. The server always works on a high priority job if one is available and high priority jobs are always run to completion once they start service. If the server is busy with a low priority job when a high priority job arrives, the low priority job is preempted (work on it is suspended) and the server then begins working on the high priority job. The server processes high priority jobs until there are no more high priority jobs to work on. Work on a suspended low priority job is then resumed where it left off. For this example we assume fixed service time requirements; high priority jobs each require 2.25 time units of service and low priority jobs each require 3.5 time units of service.

Each type of AC must have a type name. Since the AC we are modeling in this example is a server that handles two classes of jobs, we assign this type of AC the type name: "2ClassServer". A "2ClassServer" AC has two input ports "hi" and "lo", and a single output port "out". From the IG (Interconnection Graph) or HIG (Hierarchical Interconnection Graph) the AC might look similar to that shown in Figure 6. Note that we put the type name of the AC in parenthesis to indicate that "2ClassServer" is a type name and not an

instance name. All AC's of the same type have the same type name. An instance of an AC of type "2ClassServer" might resemble that shown in Figure 7 where "myServer" is the instance name of this AC of type "2ClassServer". Note that the instance name is not in parentheses, thus, for diagrams using this representation, it is easy to distinguish the instance name from the type name of an AC.

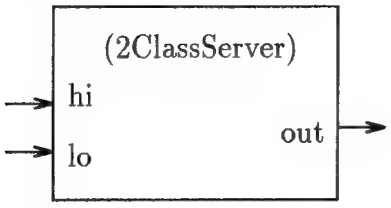

Figure 6: External view of an AC of type "2ClassServer"

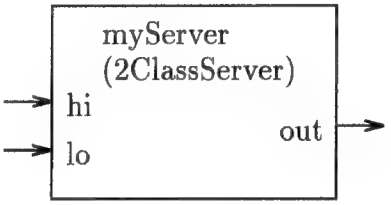

Figure 7: A specific instance of an AC of type "2ClassServer"

In an AC of type "2ClassServer", each "job" is represented by a message. Thus a job arrival is synonymous with (and represented via) a message arrival in our "2ClassServer". The priority of a job is determined by the input port on which it arrives. Thus high priority jobs arrive on input port "hi" and low priority jobs arrive on input port "lo". As jobs finish service, they are all sent out (as messages) on a single output port "out", i.e., the high and low priority jobs are no longer kept separate. If there is a requirement to be able to distinguish high and low priority jobs after they are sent to the single output port "out" upon completion of service, then it is necessary to use an attribute of the message to store the job's priority classification. (Alternatively, separate output ports could be used for each type of job, as was done with the input ports.)

We model the behavior of the AC with an HCFG. The root MCS of an HCFG tree is usually designed specifically to implement the behavior for a specific type of AC. We generally (by convention) give the root MCS the same type name as the type name of its AC. Thus, for this example, the root node of the HCFG tree is an MCS of type "2ClassServer". We model the MCS representing the root node of the HCFG tree as an MCS with four control states. We name these four control states: "I", "BL", "BH", and "P" which stand for "Idle", "Busy-Lo", "Busy-Hi", and "Preempt", respectively, as shown in Figure 8. This MCS contains no child MCS's, thus the HCFG tree for a "2ClassServer" AC is completely specified with a

single "2ClassServer" MCS which is the root (and only) node of this HCFG tree. Thus the specification of this AC by an HCFG that has no child MCS's is a CFG. (Later in this subsection we will show modifications of the "2ClassServer" MCS which have child MCS's.)

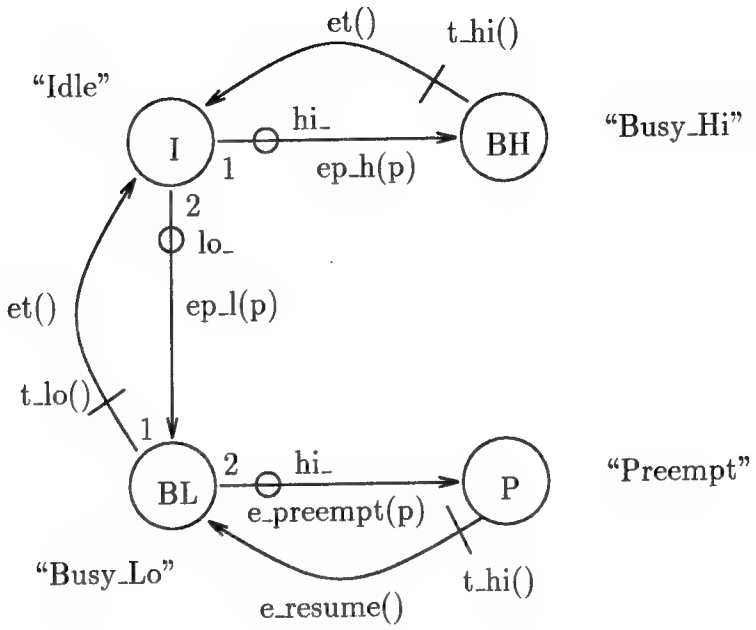

Figure 8: 2ClassServer

In an MCS each edge leaving a control state has three attributes: a priority, a condition, and an event. The priority is shown as an integer near the base of an edge. For example, in Figure 8, the edge from control state "I" to control state "BH" has a priority "1", and the edge from "I" to "BL" has a priority "2". Lower numbers represent higher priority. All edges leaving a specific control state must have unique priorities. If there is only one edge leaving a control state there is no need to explicitly specify a priority for that edge since priority is used only to select one edge from a set of edges during model execution.

Each edge has a condition attribute of one of the following three types: "time delay", "boolean expression", or "nonempty input port". We call edges which have conditions of these three types: "TimeEdges", "BoolEdges", and "PortEdges", respectively. The type of edge is indicated graphically by a symbol on the edge. In Figure 8 the edge from control state "I" to control state "BH" is a PortEdge. This is indicated graphically with a small circle near the base of the edge. A PortEdge specifies a nonempty input port condition for a specific port, so we must supply a handle (pointer) to the port on which this edge's condition is based. Thus the PortEdge from "I" to "BH" indicates that the port which is responsible for the condition on this edge is the port accessed via the port handle "hi_". A PortEdge condition is **True** if there exists an unreceived message waiting on the input port, and **False** otherwise.

TimeEdges are indicated graphically via a line perpendicular to the edge near the base of

the edge. The edge from control state "BH" to control state "I" in Figure 8 is a TimeEdge. A TimeEdge requires a function that returns a nonnegative time value Δt which is used to determine when the TimeEdge condition will become **True**. A TimeEdge condition is **False** until the local simulation time reaches $(now + \Delta t)$ and **True** thereafter (where *now* is the value of the local simulation clock at the time the edge condition is evaluated). The Δt function for the edge from "BH" to "I" in Figure 8 is "`t_hi()`" which specifies that the "2ClassServer" MCS must have a function with name `t_hi()` which returns a non-negative value for Δt. This function is called whenever a value for Δt is required to evaluate the condition attribute of the TimeEdge.

Our "2ClassServer" example does not have any BoolEdges, so we show the graphical symbol for a BoolEdge in Figure 9. A BoolEdge requires a function which returns **True** or **False**. In Figure 9 this function is "`b()`". This means that the MCS must have a function named "`b()`" which is called whenever the BoolEdge's condition needs to be evaluated.

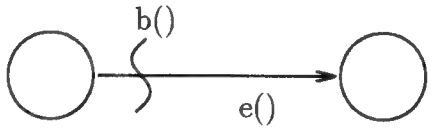

Figure 9: BoolEdge

A special case of a BoolEdge (called a "TrueEdge") in which the condition is always **True** is used with sufficient frequently that we have a special symbol for such an edge. A "TrueEdge" is shown in Figure 10. The bold "**T**" on the edge's type symbol indicates that no condition function needs to be specified for this edge since its condition is always **True**.

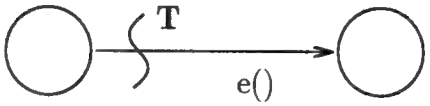

Figure 10: TrueEdge (BoolEdge with condition always **True**)

Each edge also has an event attribute which is executed when the edge is traversed. For TimeEdges and BoolEdges the event is defined by a function that takes no parameters. If events associated with the traversal of different TimeEdges and/or different BoolEdges perform the same action, then those edges may use the same event routine. Both the TimeEdge from control state "BH" to control state "I" and the TimeEdge from "BL" to "I" in Figure 8 have an event routine named "`et()`" (event for the time delay edge). This means that the "2ClassServer" MCS has a function "`et()`" which is called whenever either of these two edges are traversed during simulation execution.

Edges with associated event routines whose actions are to "do absolutely nothing" occur with such frequency that we have a special name and symbol for this kind of event. An event that does nothing is called the "*null* event" and is indicated by using the event symbol "e_{null}" on an edge whose event routine is the null event.

The events associated with edges of type PortEdge differ slightly from the events for TimeEdges and BoolEdges in that events for PortEdges are specified by a function that takes a single parameter. This parameter is a handle (pointer) to the input port on which that PortEdge's condition was based. Thus the event associated with a PortEdge knows which input port was tested during the condition evaluation for that edge. As a result of this, the event routine knows that there exists an unreceived message on that input port which is waiting to be read. In the general case, a PortEdge event will, as part of its action, receive the message from the input port, thus removing it from the input port's queue of messages. In this manner it may be possible for two or more PortEdges whose events perform the same action to use the same event routine, even if those PortEdges have conditions based on different input ports.

In addition to the graph (Figure 8), an MCS also has a set of member variables. These member variables commonly include local variables (local to the MCS in which those variables are defined, however, access to a subset of a MCS's variables may be also be explicitly exported to child MCS's of the defining MCS) and handles (pointers) to information external to the MCS (such as ports). These member variables are initialized whenever an instance of an MCS is created.

Our "2ClassServer" MCS (shown in Figure 8) has three TimeEdges, but only specifies two time delay condition functions since two of the edges use the same condition function. The time delay condition function "`t_hi()`" returns the time remaining until a high priority job completes service, and the function "`t_lo()`" returns the time remaining until the low priority job in service completes its service. Note that we always know how long a high priority job will take to complete service since high priority jobs always run to completion. However, a low priority job may be preempted by high priority jobs one or more times before it completes service, and thus we need to keep track of how far along the low priority job is in its service requirement so its service can be resumed at the point where it left off when its service was last suspended.

The "2ClassServer" MCS has six edges, but two of the edges ("BL" to "I", and "BH" to "I") have events that perform exactly the same function and thus can share an event routine (`et()`). The other edges have unique events.

We now show a variation of our original "2ClassServer" MCS (called "2ClassServerA" to distinguish it from our original "2ClassServer" MCS) in which we encapsulate a portion of our original MCS graph to form a new MCS. We draw a boundary around control state "BH" of our original MCS (Figure 8) as shown in Figure 11 and replace the encapsulated part of the graph with a child MCS named "busyHi" of type "DoHiJob" as shown in Figure 12. (Note that we represent MCS's contained within other MCS's graphically as ellipses, so it is easy to distinguish between the MCS's and control states (which are represented as circles).) The connection points of edges at the boundries of MCS's, i.e., where the edges connect to the ellipses, are called pins.

Only edges that originate on control states have the three attributes: priority, condition,

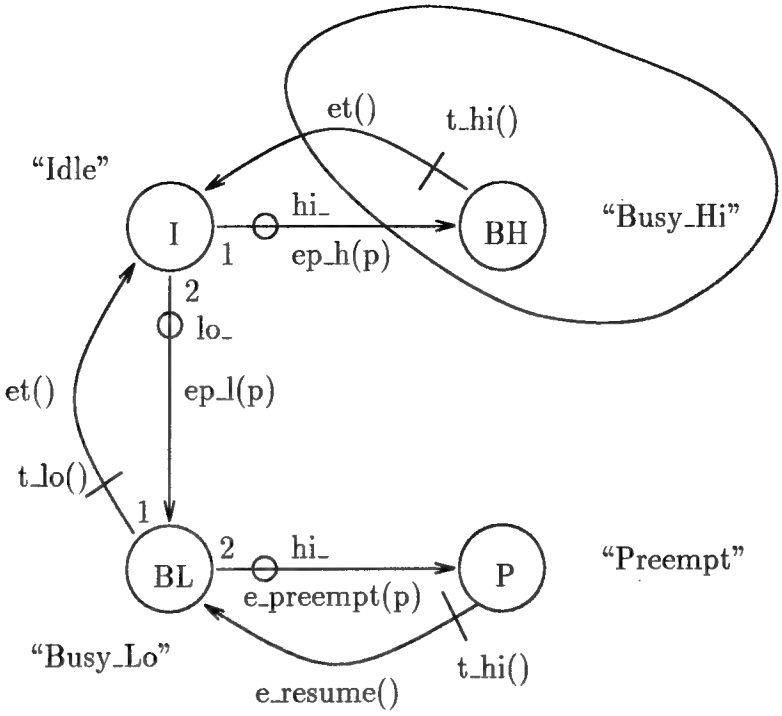

Figure 11: 2ClassServer with encapsulated node and edge

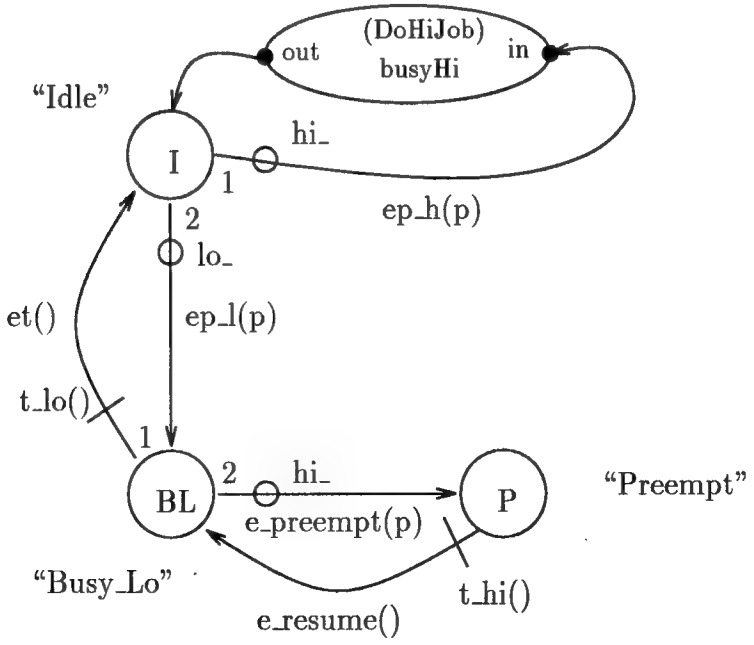

Figure 12: 2ClassServerA

and event. Thus, edges that originate on pins do not have these three attributes. When control flows over an edge (i.e., the edge is traversed) that terminates on a pin, the control flow continues through the pin and over the edge leaving the pin (each pin has exactly one edge leaving it [FS93]). This flow of control continues (possibly through a sequence of pins) until the point of control eventually reaches a control state where the point of control remains until the AC executes its next event.

In a "2ClassServerA" type MCS, if the server is idle (the "point of control" is in control state "I") when a high priority job arrives, control flows into the "busyHi" MCS (of type "DoHiJob") via the MCS's input Pin named "in". When the point of control leaves MCS "busyHi" via output pin "out", the point of control then moves to the control state "I" of the "2ClassServerA" MCS.

The next step is to specify an MCS of type "DoHiJob". The graph of a "DoHiJob" MCS is shown in Figure 13. (Note that pins are shown as circles with a cross in them.) One must also specify the member variables, the initialization function, the condition routines, and the event routines required by a "DoHiJob" MCS. Note that the point of control enters the MCS via input pin "in" and exits the MCS via output pin "out". A "DoHiJob" MCS has a single control state "BH" (BusyHi). When the point of control arrives at control state "BH" work on the current (high priority) job begins. When the service requirement for the job is completed, the point of control traverses the edge from control state: "BH" to the output pin "out".

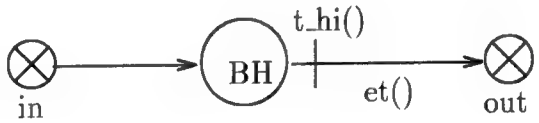

Figure 13: DoHiJob

The HCFG tree for 2ClassServerA has a root node of 2ClassServerA and has one child node "DoHiJob".

We now show another variation of a "2ClassServer" which demonstrates some of the capability which can be obtained with the use of MCS's to specify the behavior of AC's. We note that in our original "2ClassServer" MCS (Figure 8), the "point of control" always moves from control state "BH" to control state "I", and always after a fixed time delay. The same is true for control state "P" to control state "BL". The behavior of the "2ClassServer" AC in these two situations is simply a time delay of fixed duration which models the service time requirements of the high priority jobs. Since MCS's allow us to model a "partial" behavior, we can design an MCS which models a behavior consisting of a "fixed" time delay. Assuming we have a type of MCS that models a fixed time delay (we design one below), we could model our "2ClassServer" as shown in Figure 14. To maintain the distinction between our original "2ClassServer" and this new version, we will call this new type of AC a "2ClassServerB" for version "B" of our "2ClassServer". Thus the root MCS of the HCFG tree for an AC of type "2ClassServerB" is the MCS of type "2ClassServerB" shown in Figure 14. Note that

the "2ClassServerB" uses "TrueEdges" (the special type of "BoolEdge" whose condition is always **True**) from control state "FH" to control state "I" and from "FP" to "BL".

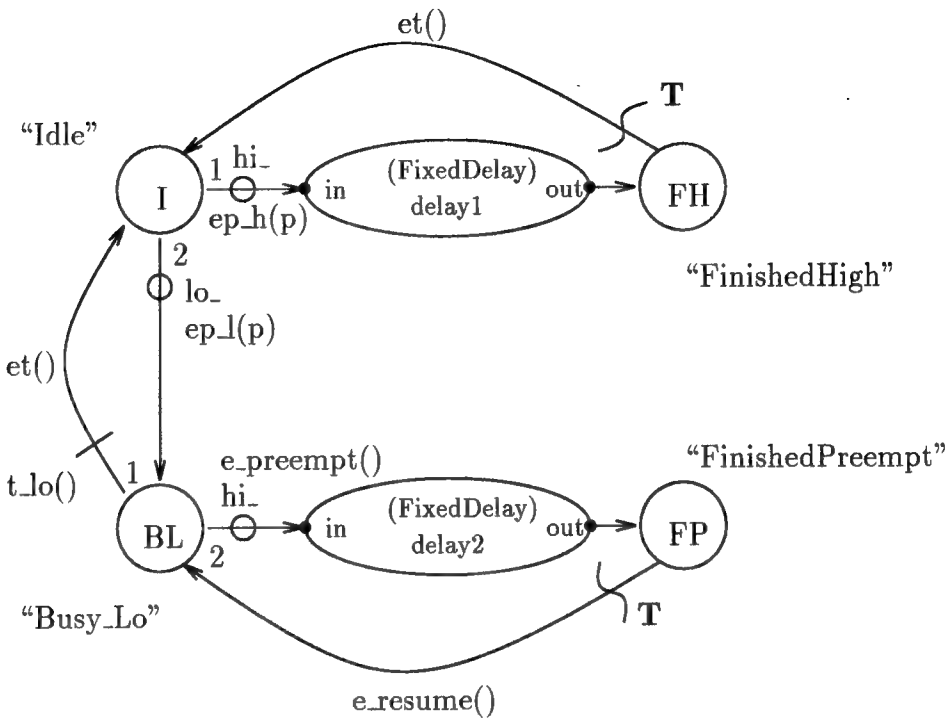

Figure 14: 2ClassServerB (Version B of 2ClassServer)

The "2ClassServerB" MCS (Figure 14) contains four control states and two MCS's. Its HCFG tree has 2ClassServerB as its root node and two child MCS's "delay1" and "delay2", both of type "FixedDelay".

Note that the rightmost two control states are new; i.e., they are different from those in Figure 8. In our original "2ClassServer" MCS, when the point of control entered control state "BH" (BusyHigh) we began service on and were busy serving a high priority job. However, in the "2ClassServerB" MCS, when the point of control enters control state "FH" (FinishedHigh), this means that we have just *finished* the service for a high priority job. The above also applies to control state "FP".

The two child MCS's, "delay1" and "delay2", are both MCS's of type "FixedDelay". (All control states and MCS's contained within a specific MCS must have unique names.) Control flow enters and exits an MCS through pins. A "FixedDelay" MCS has an input pin "in" and an output pin "out".

Our next task is to specify the definition for an MCS of type "FixedDelay". We define our "FixedDelay" MCS to have a single control state, which we call "S1" (because it is the first (and only) control **S**tate of a "FixedDelay" MCS) as shown in Figure 15. Note that the edge from pin "in" to control state "S1" has no attributes because the edge originates from

a pin. The only member variable required for our "FixedDelay" MCS is the value to use for the fixed time delay.

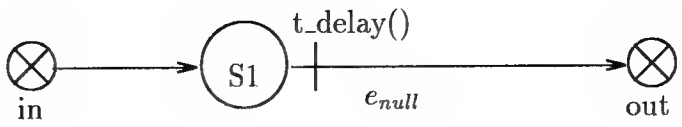

Figure 15: FixedDelay

We only have to define once what a "FixedDelay" MCS is, and then we can create as many instances of it as needed. Once we have a "FixedDelay" MCS, we can use it anywhere, and in any model where we need to model a behavior which consists of a fixed time delay. Since the "FixedDelay" MCS requires a parameter to specify the fixed time delay, we can use multiple instances of "FixedDelay" MCS's, each of which may model a fixed time delay of a different duration.

One might ask why we would want to take an MCS, such as our original "2ClassServer" MCS, and model it using other MCS's (like we did for our "2ClassServerB" MCS). One reason is that by doing so we have modularized the delay function for high priority jobs. This makes it very easy for us to change the service time requirement for high priority jobs in our model. We can now simply "plug in" any type of a "delay function" type of MCS that we have available. (We define a delay function MCS as an MCS which has a single input pin "in", a single output pin "out", and a behavior such that when the point of control enters the MCS via its input pin "in", the point of control will then exit the MCS through its output pin "out" at some later simulation time as specified by the "delay function".) We can design our own delay function MCS's and/or we can use delay function MCS's designed by others. For example, if we wanted the service time requirement for high priority jobs in a "2ClassServerB" to be a random variate sampled from an exponential distribution, we would only have to change the "`FixedDelay`" MCS to an "`ExpDelay`" MCS and provide a definition for the "ExpDelay" exponential delay function MCS. Since the behavior of the delay function type of MCS's is completely defined inside of (encapsulated within) the delay function MCS's, this type of MCS is easy to reuse.

This example is further discussed in the User's Manual [FDS95] including the "coding"of it for HI-MASS.

3 Overview of HI-MASS

The Hierarchical Modeling and Simulation System (HI-MASS) is a prototype simulation system that supports the modeling and execution of discrete event simulation models using Hierarchical Control Flow Graph (HCFG) Model specification. The Hierarchical Interconnection Graph (HIG) is specified by Visual Interactive Modeling using a Graphical User

Interface (GUI). This specification gives the components of the model and their interconnections. The behavior of each Atomic Component is specified via C++ programming language code and using a set of C++ "classes" defined in the HI-MASS system. HI-MASS uses the synchronous sequential simulation algorithm, runs on SPARC SUNs and other UNIX based systems, and can handle "reasonable size" models. HI-MASS uses the Experimental Frame concept which separates a model's definition from the set of model parameters used for specific execution runs of the model. The values for the experimental frame for a model using HI-MASS are placed in a set of files. This includes the model's initial conditions and the experimental conditions.

An executable model in HI-MASS is defined in a set of C++ source files which are compiled and linked together to form an executable program. These source files contain both the model specification and HI-MASS support code used to execute the model. The model specification consists of the HIG file, the "model" file, and the appropriate behavior specification files for each type of Atomic Component. The executable model initializes its internal state using information from the Experimental Frame files prior to commencing each simulation run. Figure 16 gives an overview of HI-MASS and of this process.

Limited Help and Trace capabilities exist in HI-MASS. Command line options are used to specify parameters for each simulation run. Options include: (1) what trace information is desired, (2) whether to run the model by single stepping events or to run to completion, and (3) which experimental frame files are to be used to set the initial conditions and parameter values for this run.

A User's Manual [FDS95] consisting of 82 pages has been developed for HI-MASS. The manual contains different examples to illustrate the use of HI-MASS. A user of HI-MASS needs to have familiarity with modeling using the HCFG model specification, C++ [Str91], and the UNIX operating environment including its tools.

HI-MASS is based on object oriented design, is programmed primarily in C++, uses the InterViews toolkit for the GUI, and consists of approximately 25,000 lines of code and 60 C++ classes in the simulator.

The Graphical User Interface (GUI) uses two types of graphical objects to specify the Hierarchical Inconnection Graph (HIG) via Visual Interactive Modeling. Basic objects directly represent discrete elements of a HIG and include such objects as atomic components, coupled components, channels, and ports. Helper objects do not directly correspond to any element of a HIG but are necessary to specify a HIG graphically. These include objects such as joints which allow channels to change directions and connection boxes which allow for complex connection of components. The basic objects also include two types of compound objects: Component Arrays which consist of an indexed homogeneous array of components and Multichannels which consist of an indexed bundle of channels.

It is easy and straightforward to specify a HIG via the GUI. One places objects onto the screen by "clicking" on and using the appropriate objects from the "tool buttons." Where appropriate, a box "pops up" to enter the name of the object (such as when adding a component). When channels are added, ports are automatically added to the appropriate components and unique port names (numbers) are assigned to those newly created ports. Editing capabilities such as move, delete, copy, paste, clone, etc. are available in the GUI.

The GUI was developed using the C++ programming language and the InterViews user

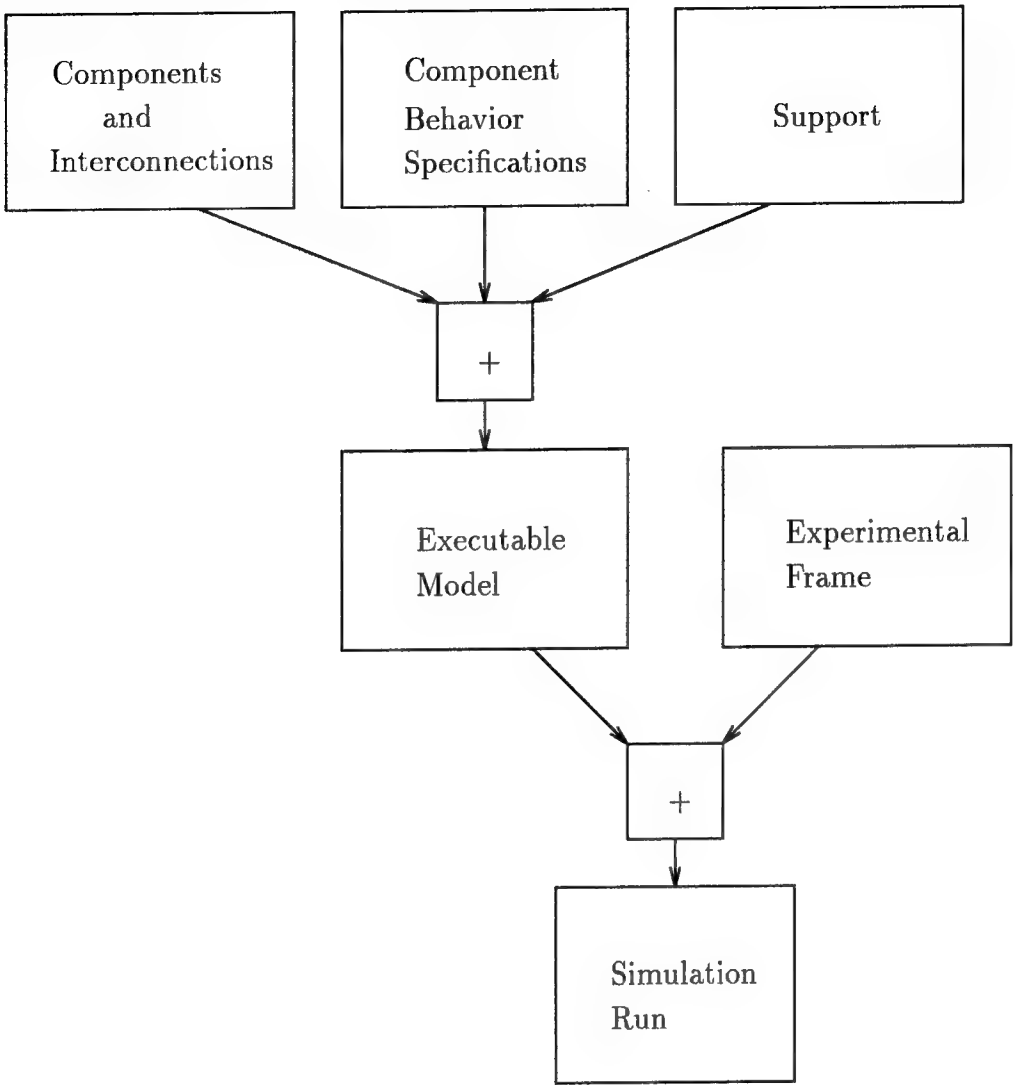

Figure 16: HI-MASS Overview

interface toolkit. Two sets of files are used for each HIG specified. One set is for the graphics information used by the GUI and the other set is for the specification of the HIG for the simulator.

The behavior of each Atomic Component (AC) in HI-MASS is specified in text via C++. Two basic classes provided by HI-MASS are used extensively. One is the class "AC" and the other is class "MCS" (Macro Control State). The specification of the behavior of each AC requires the use of the class AC once and the class MCS at least once. A "top" level MCS is needed for each AC and an additional class MCS is required for each additonal type of MCS used in the specification of the AC. In addition to using these classes to obtain files for specification of an AC, C++ code for specifying the events, time delay functions, etc. that are used in the MCS's are also required.

To specify a simulation model in HI-MASS, one needs to first specify a complete model. This includes using the class "Model" provided by HI-MASS and an Interconnection Graph (IG) file. The IG file is obtained by converting a HIG specified using the GUI into an IG where the components and interconnections are specified via C++ source code. Also needed are two files for the experimental frame. One file specifies the initial control state for each AC in the model, and the other file specifies the initial values of the model variables. All of the C++ source files used in the model specification (including the files for the AC's and the MCS's) are compiled and then linked together with the HI-MASS (and associated) libraries. These libraries contain the simulation execution algorithm, the HI-MASS class definitions, and other items such as the pseudo-random number generators. The output file produced by the linker is the executable model. The executable model is then run with appropriate command line options (including the options which specify the experimental frame files to use for a specific simulation run). See Figure 17 for an overview.

To execute a model, two modes are provided. One is to execute the model one event at a time (to aid in debugging). The other mode is to run the model until termination occurs. Different types of trace information can be specified to be collected: (1) control flow, (2) message traffic, (3) data associated with events, (4) data not associated with events, and (5) event list operations.

4 Evaluations and Lessons Learned

We give in this section the lessons learned in this research, and evaluations of HCFG models as a way of specifying hierarchical models for discrete event simulation and of HI-MASS. Each of the two evaluations is given in a separate subsection and they include most of the lessons learned during this research effort. The most important lessons learned are presented next.

We found that considerable effort was required to design and develop a prototype simulation system that has hierarchical capabilities. HI-MASS provides for both types of hierarchical capabilities (coupled components in the HIG and Macro Control States in the HCFG's) in HCFG models. Each of these two independent types of hierarchical methods has modularity of entities which includes encapsulation and locality. We designed and implemented a Graphical User Interface (GUI) using InterViews for the specification of the HIG via Visual Interactive Modeling. We found that developing a GUI that has hierarchical specification capability requires considerable effort. A complex software system is required to support the behavior specification of AC's using HCFG's with MCS's. Considerable effort (and much more effort than needed for the development of the GUI for the HIG specification) was required for designing and developing this software which is for specifying the behavior of AC's only by text! The use of the object oriented language C++ helped in this process.

We found that Visual Interactive Modeling is preferable by far over specification by text and especially so for hierarchical modeling. The specification of a HIG in HI-MASS using the GUI is straightforward and requires (1) less time to develop a HIG specification and (2) less knowledge of the underlying software system than if the HIG specification is to be entered using only textual means. We believe it would significantly aid the specification of the HCFG's of each AC if Visual Interactive Modeling was available to aid in this specification.

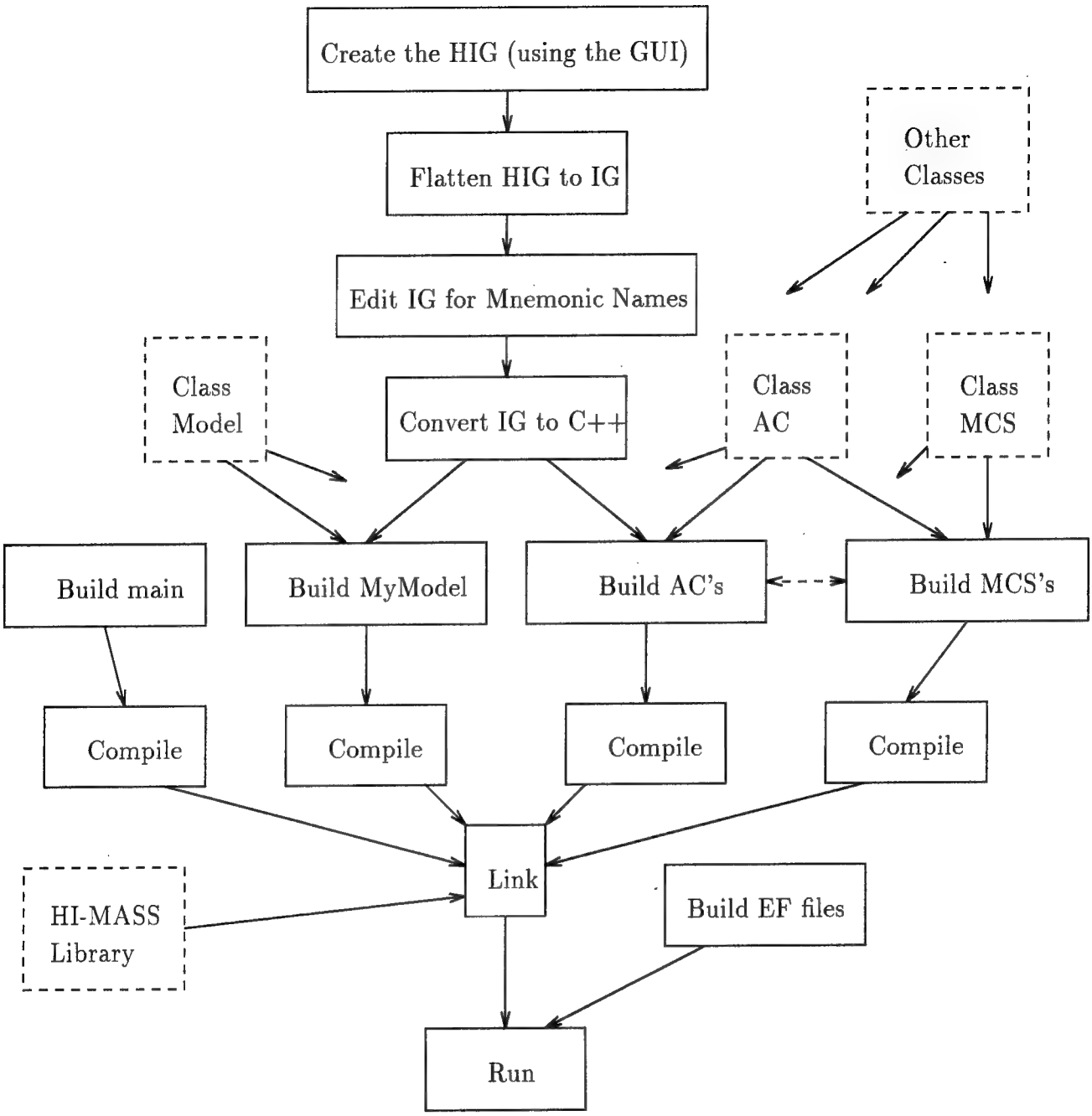

Figure 17: The general process for a HI-MASS model

Developing this additional capability into HI-MASS (or a similar simulation system) would require significant effort but is feasible and desirable to do.

Other important lessons learned are that the hierarchical modeling capabilities provided by HCFG models are extremely powerful, both types of hierarchical capability in HCFG models are important, and that further development of HCFG models is needed to achieve

their full capability. This is discussed in some detail in the subsection on evaluation of HCFG models for hierarchical modeling.

4.1 Evaluation of HCFG Models for Hierarchical Modeling

The two forms of hierarchical modeling (coupled components and macro control states) provided in Hierarchical Control Flow Graph (HCFG) models give an extremely powerful way to model systems, especially large-scale systems and/or systems that have similar components. The ability to have coupled and atomic components allows a modeler to divide a system into as many components and subcomponents as preferred for modeling and to have different levels of fidelity for them. The recursive nature of coupled components provide a powerful and straightforward way to model. In addition, these components are encapsulated and reusable.

The other form of hierarchical capability is new and allows a modeler to recursively partition and then specify the behavior of an atomic component. The HCFG behavior specification is constructed using (simple) control states and macro control states (MCS's). The use of control states gives a new "world view" for specifying the behavior of systems/models. (The current world views are event-scheduling, activity-scanning, and process-interaction (which includes the transaction view).) In the HCFG paradigm, a modeler can recursively specify the behavior of atomic components by using MCS's. MCS's are encapsulated and have significant reuse capability.

We conducted modeling of several systems using the two types of hierarchical capability in HCFG models. The capability of "coupling components" is as one would expect: powerful and straightforward. However, scaleability would greatly aid in specifying a HIG, i.e., the ability to specify different sizes of the HIG's "entities." In HI-MASS, we provide "arrays" of components and "multichannels" between components. This is a static specification capability that was not in the original HIG paradigm. Scaleability of the HIG needs to be investigated both theoretically and practically, including the capability to specify the size of the entities through an "experimental frame."

We found the capability of specifying the behavior of atomic components via HCFG's using (simple) control states and macro control states (MCSs) to be extremely powerful. However, we also found that there is much to be learned about this new way of specifying behavior and that additional capabilities in them are needed. For example, there are numerous ways to specify the behavior of atomic components via MCS's in the HCFG paradigm. These different ways occur because of the choices available to the modeler. One consideration is how to choose an appropriate granularity with which to partition the behavioral state space of the ACs. The use of a coarse partitioning results in fewer, but generally more complex MCS's, and a finer partitioning results in more, but relatively simpler MCS's. Another consideration is how to best exploit the recursive decomposition capability of HCFG's. The modeler must chose whether to model using a shallow hierarchy, or whether to model using a deep hierarchy. The deeper hierarchy approach tends to have simpler MCS's but more complex MCS interfaces as the sharing of resources between parent and child MCS's tends to be higher than that between sibling MCS's. A third consideration is how to best divide the complexity of each MCS between its "graph" and its "event routines". A MCS with a

large number of control states may have simple event routines whereas the same behavior can be specified using a MCS with fewer control states with more complex event routines. The tradeoffs among these alternative ways of modeling behavior using the HCFG paradigm is not well understood and thus needs research.

Research is also needed in the area of data collection and analysis for HCFG Models. Ideally, a modeler would like to be able to specify the data collection required for a particular simulation study independently of the behavior being modeled. With this type of capability it would be possible to more easily reuse the same behavioral specification for simulation studies with different goals but which involve the same or similar model elements.

Scaleability is a capability needed in HCFG's. This is extremely important when similar multiple edges are used between the same set of control states. Scaleability is needed to provide for an "array of MCS's." In addition, there is the interaction of scaling between components and behavior of components (HCFG's). Consider, for example, the case where a modeler uses an array of components to send messages to a single atomic component, and this atomic component is to receive these messages on edges originated or terminating on a single control state. This requires a set of edges for that control state set where the number of edges is determined by the size of the array of components. How to do this simply and how to specify the size of both entities through the use of an experimental frame needs research.

Another area which needs additional research regarding MCS's is how to best support the reuse of MCS's. A library and documentation support system is needed in order to effectively reuse MCS's as we must have some type of repository for an existing set of MCS's and we must also have sufficient documentation to understand the behaviors, interfaces, and limitations of these MCS's. Simpler, more generic behavior specification MCS's would also tend to be more easily reused than application specific MCS's, but the development of application specific libraries of MCS's should also be investigated. Another capability that we feel would significantly enhance reusability is the development of parameterized (and scalable) MCS's. (An example of a parameterized MCS is the "FixedDelay" MCS in Figure 15.) Our preliminary investigations have found that parameterized MCS's are almost a requirement for effective reuse. "Object oriented" inheritance is another issue that needs to be addressed with respect to reusability. Some preliminary work relating to the creation of new types of MCS's by inheriting from existing types of MCS's appears promising. This "inheritance" method of reuse which supports the "is-a-kind-of" relationship between the derived MCS and the base MCS's has also shown some usefulness in the separation of behavior specification and data collection. For example, suppose that we have a MCS of type "Fred" that only specifies a behavior. Then we could create a new MCS of type "Fred-with-data-collection" by simply inheriting the behavior specification of "Fred" and adding only the additional data collection part of the specification. Those properties of MCS's which best support these methods of reuse need to be explored in order to develop a set of guidelines for the MCS designer.

Probably much could be learned with respect to modeling using HCFG's and MCS's by studying their relationship to the other world views used in discrete event simulation modeling. This might even lead to specifying a set of MCS's that could be used similarly to the transaction version (e.g., GPSS) of the process-interaction world view.

An important observation that we made in modeling systems using HCFG models was

the tremendous flexibility that exists in modeling using HCFG models. One can use many or a few components, use messages in different ways, use many or few MCS's in atomic components, and have simple or complex atomic components resulting in simple or complex behavior (and thus simple or complex HCFG's). A related and important observation we made in specifying hierarchical models was a tendency to model systems using many simple AC's in order to have simple behavior specifications (HCFG's) instead of using fewer AC's that were more complex requiring more complex HCFG's. It appears to be easier to model using many simple AC's than fewer and more complex AC's. This is an important issue to research. Related to this issue is how would one model if there existed a library of "complex" AC's and complex MCS's that are easy to use (or reuse) in modeling, thus eliminating the need for the modeler to develop their specification. We note that having fewer AC's in a model gives faster execution time when using the sequential synchronous simulation algorithm, which is the algorithm in HI-MASS.

Another observation of interest is how does the current approach of having HCFG models based upon CFG models in order to have parallel/distributed simulation algorithms affect HCFG model specification. If we had HCFG models for only sequential synchronous computation, it might be easier to model by providing easier or more powerful ways of modeling, such as in scaling. (If sequential synchronous computation is used, "automatic" lookahead is not needed.)

Another topic of interest is developing sets of MCS's and AC's for different types of problems, such as queueing systems; or different types of application domains, such as communication systems, computer systems, or manufacturing systems. A library of appropriate MCS's and AC's could make modeling quite simple and fast for these types of systems. A GUI could perhaps be developed providing for Visual Interactive Modeling.

4.2 Evaluation of HI-MASS

The Hierarchical Modeling and Simulation System (HI-MASS) is a prototype simulation system that supports the modeling and execution of discrete event simulation models using HCFG Model specification. Although HI-MASS was developed using a small amount of resources and has significant areas in which improvements can be made, we find the current version to be quite powerful. To use HI-MASS effectively, a modeler must be familiar with HCFG modeling, C++ programming, and the Unix operating system environment and its associated software tools.

Specifying a HIG via Visual Interactive Modeling using the HI-MASS GUI is straightforward. The specification of AC's and their behaviors in HI-MASS is via text using the C++ programming language and a set of predefined (in HI-MASS) C++ classes. This requires a modeler to be familiar with the concepts involved in the modeling of an AC's behavior using HCFG's and also how those HCFG modeling concepts are represented in HI-MASS. A number of improvements to the current version of HI-MASS that can make the specification of AC's and their behaviors easier are described below. To specify a "model" in HI-MASS and execute it requires significant knowledge of HI-MASS, C++, and the Unix operating environment and its tools. This can also be simplified.

HI-MASS was developed to satisfy the objective of the contract, which was to demonstrate

the feasibility of and evaluate the useability of HCFG Models. It satisfied these purposes. The current version of HI-MASS can be used to simulate models of a reasonable size. There is a User's Manual for HI-MASS which includes several examples. HI-MASS is also a good prototype with which to experiment with and learn more about HCFG model specification.

Important areas of improvements that can be made to the current version of HI-MASS are the following:

- Provide displays of HIG and HCFG Trees
- Provide the capability to specify parameters in the HIG specification via Experimental Frame
- Provide annotations of the HIG via Visual Interactive Modeling
- Increase the Trace and Help capabilities
- "Automate" the generation of files for Modeling and Simulation Execution
- Develop a set of "basic" MCS's and AC's
- Develop a library and library management system for MCS's, AC's, coupled components, and models
- Develop a Visual Interaction Modeling capability to aid in specification of AC's and MCS's
- Develop animation of the Point of Control (POC) as it moves within an HCFG of an AC (would be shown on CFG or MCS's depending on approach used.)
- Provide data analysis capability

Other areas of improvement of HI-MASS include the improvements suggested for HCFG models under their evaluation in Subsection 4.1 (such as scaling), and to provide other simulation computation (parallel/distributed) algorithms in HI-MASS in addition to the sequential synchronous simulation algorithm used in the current version of HI-MASS.

5 Recommendations and Summary

We first present the recommendations and then the summary. In presenting the recommendations, we divide them into additional investigation and development of HCFG models and further development of HI-MASS. Furthermore, we divide the recommendations for HI-MASS into three different levels (or groups) depending on effort required. Most of these recommendations follow from the evaluations and lessons learned presented in Section 4.

The recommendations are the following:

- Model more systems to gain further insight into modeling using this new world view of HCFG models and to determine what additional capabilities are needed in this specification.

- Further develop the capability of HCFG Models
  - Scaleability of HIG, HCFG, and their interactions
  - Basic Set of AC's and MCS's
  - Study to gain a better understanding of the tradeoffs involved in modeling styles that use a larger number of relatively simple AC's versus those that use fewer but more complex AC's
  - Study for better understanding of hierarchical relationships of MCS's with respect to modeling
  - Investigate relationships of HCFG's world view to other world views used in discrete event simulation
  - Investigate HCFG specifications required for sequential versus parallel/distributed simulation computation.
  - Investigate executing HIG directly, instead of flattening as used in HI-MASS
  - Further explore the execution of HCFG's hierarchically versus flattening first
- Develop a Library Management System for MCS's, AC's, Coupled Components, and Models
- Modifications to Increase HI-MASS Capabilities
  - Level I (No research needed; Aid in ease of usage of current version of HI-MASS)
    - Capability to display HIG and HCFG Trees
    - Add annotation capability to HIG specification via GUI
    - "Automate" the straightforward portions of Model and Simulation Construction
    - Increase Trace and Help Capabilities
    - Provide data collection and analysis capabilities
    - Develop GUI for MCS's
    - Optimize the simulator code
  - Level II (Limited research required; add capabilities to current version of HI-MASS)
    - Develop a set of "Basic" AC's and MCS's
    - Provide limited scaleability of "entities"
    - Develop menu driven "automation" of model and simulation construction and execution
    - Develop sets of MCS's and AC's for application and problem domains and implement via GUI for Visual Interactive Modeling
    - Library Management System for MCS's, AC's, etc.
  - Level III (Research and Experimentation needed; add capability to HI-MASS; results in a new version of HI-MASS)
    - Provide Scaleability of HIG, HCFG's, and their interactions

- Provide Visual Interactive Modeling of AC's
- Provide animation of Point of Control (POC) in HCFG's
- Provide animation of message traffic within HIG
- Implement parallel/distributed simulation computation algorithms

We now summarize this research effort. The objectives of the contract were satisfied. We demonstrated the feasibility of and evaluated the useability of HCFG Models to specify hierarchical models for discrete event simulation. We developed a prototype simulation system called HI-MASS that uses the HCFG Model specification. We found that the HCFG Model specification is extremely powerful, flexible, allows for reuse, and should be further developed. Suggestions for further research and development of HCFG Models were made. We show in developing HI-MASS that a simulation system that supports HCFG Model specification can be developed for workstations, that visual interactive modeling is highly desirable for hierarchical modeling, and found significant effort is required to develop a simulation system to support hierarchical modeling. We also determined a number of ways that the current version of HI-MASS can be improved and presented them. Recommendations for future versions of HI-MASS were made, including providing for parallel/distributed simulation computation.

References

[CS89] B.A. Cota and R.G. Sargent. Automatic lookahead computation for conservative distributed simulation. CASE Center Technical Report 8916, Syracuse University, December 1989.

[CS90a] B.A. Cota and R.G. Sargent. Control flow graphs: A method of model representation for parallel discrete event simulation. CASE Center Technical Report 9026, Syracuse University, December 1990.

[CS90b] B.A. Cota and R.G. Sargent. A framework for automatic lookahead computation in conservative distributed simulations. In D. Nicol, editor, *Distributed Simulation*, pages 56–59, 1990. The Society for Computer Simulation.

[CS90c] B.A. Cota and R.G. Sargent. Simulation algorithms for control flow graphs. CASE Center Technical Report 9023, Syracuse University, November 1990.

[CS90d] B.A. Cota and R.G. Sargent. Simultaneous events and distributed simulation. In O. Balci, R.P. Sadowski, and R.E. Nance, editors, *Proceedings of the 1990 Winter Simulation Conference*, pages 436–440, 1990.

[CS92] B.A. Cota and R.G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, 2(2):109–129, April 1992.

[FS93] D.G. Fritz and R.G. Sargent. Hierarchical control flow graphs. CASE Center Technical Report 9323, Syracuse University, December 1993.

[FDS95] D.G. Fritz, T. Daum, and R.G. Sargent. User's Manual for HI-MASS. Simulation Research Group, 439 Link Hall, Syracuse University, February 1995.

[Fuj90] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[Jef85] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):198–206, July 1985.

[Mis86] J. Misra. Distributed-discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[Rey82] P.F. Reynolds. A shared resource algorithm for distributed simulation. In *Proceedings of the Ninth Annual International Computer Architecture Conference*, pages 259–266, 1982.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison–Wesley, second edition, 1991.

[Zei84] B.P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* Academic Press, 1984.

☆U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20129

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.